Pruning of Genetic Programming Trees Using Permutation Tests

Peter Rockett
Department of Electronic and Electrical Engineering
University of Sheffield
Portobello Centre
Pitt Street
Sheffield S1 4ET
UK

Technical Report No. CR2018-1

9 July 2018

Abstract

We present a novel approach based on statistical permutation tests for pruning redundant subtrees from genetic programming (GP) trees. We observe that over a range of regression problems, median tree sizes are reduced by around 20% largely independent of test function, and that while some large subtrees are removed, the median pruned subtree comprises just three nodes; most take the form of an exact algebraic simplification. Our statistically-based pruning technique has allowed us to explore the hypothesis that a given subtree can be replaced with a constant if this substitution results in no statistical change to the behaviour of the parent tree—what we term approximate simplification. In the eventuality, we infer that $\gtrsim$95% of the pruned subtrees are the result of algebraic simplifications, which provides some practical insight into the scope of removing redundancies in GP trees.

1 Introduction

It has long been accepted that genetic programming (GP) produces trees that contain substantial amounts of redundancy [1–5]. The objections to this are well rehearsed: the tree evaluation time is increased, and the redundant subtrees may obscure human interpretation of the evolved solution. It is therefore desirable to remove as much of this redundant material as possible. Although manual tree simplification appears to have been used in the past, automated tree simplification is much preferred but is very challenging [6]; Naoki et al. [7] point out that canonical simplification is not Turing computable. Work on the simplification of GP trees has been reviewed by Kinzett et al. [8].

Nordin et al. [9] have defined a taxonomy of *introns*—broadly, the redundant code fragments that make no contribution to the tree's overall fitness; some of these are noted to depend on particular test cases [5, 9]. Jackson [5] has addressed removal of a subset of introns, *dormant nodes*, that are never executed.

Zhang and co-workers [2, 6] have explored the use of hashing to simplify trees both at the end of a run as well as during the evolutionary run. These authors found, unsurprisingly, that simplification reduced tree sizes although the effect on test performance was not examined with any formal statistical procedure and does appear to have been resolved. Interestingly, Zhang et al. concluded that their initial hypothesis that frequent simplification would reduce genetic diversity and therefore hinder search proved unfounded; they did, however, find evidence against applying simplification at every generation.

In addition to exploring a hash table based approach to algebraic simplification, Zhang and co-workers [10, 11] have also examined approximate numerical simplification methods based on the local effect of a subtree. Kinzett et al. [10] replaced a subtree by the average of its output over the training set if the range of its outputs fell below a user-defined threshold. Song et al. [11] pruned trees by comparing the output of a binary node with its two inputs and replacing that node with either child if it gave the same value as the binary output within a threshold. The drawbacks with both these contributions are that: i) they involve local operations that ignore the effect of an edit higher in the tree, and ii) both rely on setting user-defined thresholds for which there appears to be no principled method other than trial-and-error. The shortcoming of ignoring the propagated effects of changes higher in the tree was examined by Johnston et al. [12] although they too used a two-stage process that relies on a user-defined threshold to gauge the acceptability (or otherwise) of a proposed simplification. Further, these authors only considered the propagated influence of a proposed simplification one or two levels up the tree.

Naoki et al. [7] have applied two simplification methods in tandem: one based on a set of rewriting rules derived from normal algebra (e.g. $x \times 1 \to x$), and a second using an approximate numerical method that compares the output of every subtree in a parent with the outputs of a small library of simple trees; if a library tree provides an 'equivalent' output to the subtree under examination and is simpler, the library tree is substituted. This second method was found to be particularly effective in reducing tree size, but suffers from a number of drawbacks: i) the method again relies on a user-defined threshold to gauge similarity, and ii) definition of the comparison library is domain-specific and potentially problematic.

Recently, Helmuth et al. [13] have exploited the properties of a specific GP system for which randomly excluding fragments still results in a legal program. Over a series of software synthesis problems, these authors improved performance over a test set by repeatedly proposing the removal of randomly-selected program fragments in a Monte-Carlo fashion, and accepting those proposals that improved the test set score. We have reservations, however, about the claims of improving program generalisation: unless we have misinterpreted the work, all simplification decisions appear to have been made using a single test set (as opposed to an independent *validation* set—see [14, p.222]), which probably results in over-specialisation on the test set, but with no guaranteed improvement in program generalisation.

Although the hash table based algebraic simplification of Zhang et al. and other rule-based approaches are interesting contributions, they too suffer from a number of drawbacks. Principally, algebraic simplification is limited in scope and does not accommodate 'effective' redundancy [12]. For example, consider the expression $x+\epsilon$ where x is of the order of unity and ϵ is, say, 10^{-32}. To all intents and purposes, the result of this addition operation is x since it is highly unlikely that including the factor of 10^{-32} will make any significant difference to the tree's prediction. Algebraic simplification, however, would fail to recognize the $x+\epsilon$ fragment as redundant because—in strict algebraic terms—it is not, and this has led researchers to employ approximate numerical methods. Many numerical simplification approaches have the over-riding disadvantage of requiring user-defined thresholds to gauge whether a candidate subtree has 'no' effect and can thus be pruned. In fact, the combined approach of Johnston et al. [12] requires the setting of six user-defined thresholds.

In this report we adopt an approach akin to numerical simplification [10]: we regard a subtree as redundant if its removal results in *no statistically significant change* in the output of the parent tree: our approach thus provides a principled method for accepting/rejecting a pruning proposal and avoids the need for user-defined parameters. We explore the use of statistical permutation tests to determine if a selected subtree can be replaced with a constant of value equal to the mean of the subtree's output over the data set; replacement by a constant node has also been followed by [10]. Other types of redundancy can be readily addressed by our scheme, but for the first report of this novel application of permutation testing to GP tree pruning, we restrict attention to exploring this cause of tree redundancy. The key advantage over previous work on simplification is:

- Whether or not to accept a pruning proposal is judged *statistically* on the probability that a given pruning operation will change the output of the tree. This gives principled grounds for pruning decisions based on the long-term probability of erroneous pruning. We thus avoid arbitrary, user-defined thresholds.

In this initial report we limit our approach to pruning the 'best' tree produced by a conventional GP run. In Section 2, we describe our GP tree pruning approach, and in Section 3 we outline the necessary background on permutation tests. The experimental methodology employed is described in Section 4 and results are presented in Section 5. We conclude the paper with a discussion and areas for future work in Section 6 and Section 7.

2 The Pruning Approach

Considering the simple example tree representation in Figure 1 which implements the quite general mapping $y = f(\mathbf{x})$ where the vector $\mathbf{x} = (x_1\, x_2)^T$.

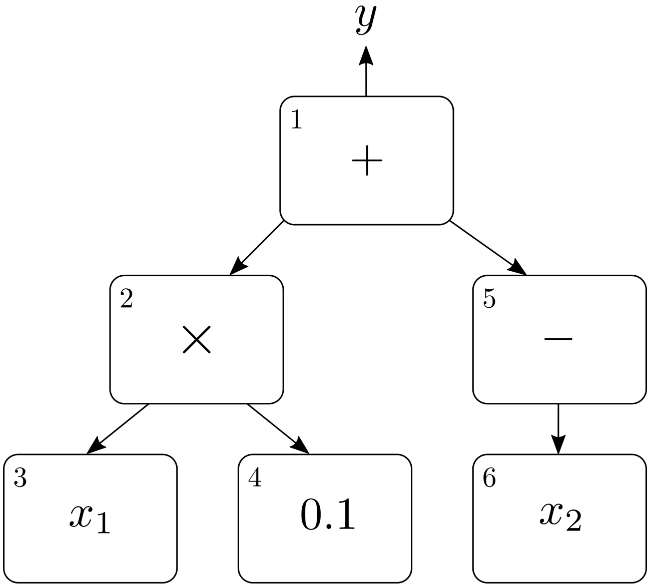


Figure 1: Example of a simple GP tree. The node numbers are shown in the top left corners of the nodes.

The normal recursive evaluation begins with the root node '1' and proceeds by the following sequence. Firstly, the subtree rooted by node '2' is evaluated. Since '2' is not a terminal node, '3' is evaluated. Since this node is a terminal, this phase of the recursive evaluation stops and the value of x_1 is returned to '2'. At this, '4' is evaluated and since this is a constant (and therefore a terminal), a value of 0.1 is returned to '2'. At this point, the numerical result from the subtree rooted at '2' can be evaluated and returned to '1'. Now, '5' is evaluated and subsequently '6' . Since node '6' is a terminal node, the value of x_2 is returned to '5' and thence the result from the subtree rooted at '5' is returned to '1'. At this point, the results from the two subtrees of '1' are combined (using the binary operation specified in '1') and the final evaluation result from the tree returned by the root node.

We consider the case of a subtree that produces the mapping $g(\mathbf{x})$ and denote $c = \langle g(\mathbf{x}) \rangle$ where $\langle\,\rangle$ denotes the expectation taken over a dataset. If replacing the given subtree by a constant node returning c does not produce a statistically significant change in the output of the parent tree then we can effect a simplification of the tree. We refer to this type of pruning as *constant subtree* (CST) pruning.

In order to implement the constant subtree pruning strategy described in this work, the basic tree structure in Figure 1 together with its recursive evaluation has to be modified. Firstly, we add an internal state variable to every node type. Thus:

$$\text{NodeState} \in \{\text{Untested}, \text{Testing}, \text{Failed}, \text{Pruned}\}$$

where the different state values have the following interpretation:

- 'Untested' denotes that the subtree rooted at that node has not yet been considered for pruning.
- 'Testing' denotes that a node is currently in the process of being evaluated for pruning.
- 'Failed' indicates that the subtree rooted at this node has been considered for pruning, but that the pruning proposal has been rejected.
- 'Pruned' indicates that a node and its subtree have previously been considered for pruning, and the pruning was judged to make no statistical difference to the tree output; in other words, the node and its subtree are redundant and can be replaced by the expectation of the subtree output over the dataset.

In addition, we also define within each node (apart from constants) a single floating-point number Z_n, where n indexes the node, and which serves the dual purpose of: i) holding the expectation over the dataset of the subtree output, and ii) acting as a running accumulator during computation of the sum of the subtree outputs prior to normalisation to form the expectation.

In order to perform a set of pruning tests, we require a given tree to be set up. This requires three functions:

`SetSumsToZero`. The function recursively descends the tree, setting to zero the floating-point number Z_n in each node.

`TreeEvaluateAccumulate`. This function takes the input vector $\mathbf{x}$ as an argument. In essence, it is externally identical to the normal function for recursively evaluating a tree described above, and returns the result of the mapping $y = f(\mathbf{x})$ implemented by the tree. In addition—with the exception of constant nodes—this function accumulates the evaluated results of every subtree for pattern $\mathbf{x}$ to a running total Z_n maintained within the subtree's root node.

`NormaliseMeans`. This function takes as its only argument the number of patterns in the dataset; it recursively descends the tree, and at every node divides the accumulated total output of every subtree by the number of patterns in the training set, thus setting Z_n equal to the average subtree output for every node.

On initialisation of the process of pruning, the above functions need to be called in the following order:

1. `SetSumsToZero`

2. `TreeEvaluateAccumulate` repeatedly within a `for`-loop over every record in the dataset
3. `NormaliseMeans` is called finally so that each node contains the average of its subtree's output

Note that since `TreeEvaluateAccumulate` returns the tree outputs $y_i = f(\mathbf{x}_i) \quad \forall i \in [1 \ldots N]$ where N is the number of data, these individual outputs are cached in an array D_u, where the 'u' subscript denotes unpruned, for subsequent use in permutation testing.

To investigate the set of hypotheses for pruning subtrees, the following functions are needed:

`SetAllToUntested`. This function recursively descends the tree and sets the internal state of every node to 'Untested', with the exception of constant nodes, the states of which are set to 'Failed'. Note that a constant node cannot be pruned without producing a broken tree.

`SetToTesting`. This function recursively descends the tree until it visits a node for which the state is 'Untested'. At this point it sets the state of that node to 'Testing' and returns a value of true. If the function is unable to locate any node in an 'Untested' state, returns a value of false indicating that all possible nodes in the have been examined for pruning. Note, only one node/subtree at a time is considered for pruning.

`TreeEvaluatePrune`. This function takes an input vector $\mathbf{x}$ as argument. The function recursively evaluates the tree in exactly the same way as the previous tree evaluation function with the following exceptions—if the recursive evaluation enters a node for which the state is:

1. 'Testing' or 'Pruned', the value of the average output of that subtree (Z_n) is returned.
2. 'Failed' or 'Untested', recursive evaluation is continued as normal.

The values returned by the outermost call of `TreeEvaluatePrune` are stored in an array D_p for subsequent use in the permutation test, where the 'p' subscript denotes 'pruned'. These are the tree responses conditioned on pruning the single subtree identified by its 'Testing' state.

`SetTestingToFailed`. This is called after a permutation test to indicate that the just-tested subtree cannot be removed without changing the output of the tree. The function recursively descends the tree to locate the node that has been set to 'Testing'. It then sets this node's state to 'Failed'. (If the function is unable to locate any node state of 'Testing' it indicates a processing error.)

`SetTestingToPruned`. This function recursively descends the tree to locate the node which has been set to 'Testing'. It sets this node's state to 'Pruned' to indicate that this subtree can be removed without affecting the predictions of the tree. This function is only called after a permutation test has indicated that the just-tested subtree can be pruned. If the function is unable to locate any node state of 'Testing' it indicates a processing error.

The pruning process proceeds in the following sequence:

1. `SetAllToUntested` is called to set all node states to 'Untested'.
2. The function `SetToTesting` is called to set the first node state to 'Testing'.
3. A `while`-loop is executed while ever the function `SetToTesting` returns true, indicating that a node in the tree has been located which is still in an 'Untested' state. Within the while-loop:
   (a) `TreeEvaluatePrune(` $\mathbf{x}_i$`)` is executed for all records in the dataset $\mathbf{x}_i \in [1 \ldots N]$, where N is the number of items in the dataset. The output of `TreeEvaluatePrune` is stored in array D_p.
   (b) A permutation test—see Section 3—is carried out using the cached data in the arrays D_u and D_p. If the null hypothesis explored by the permutation test is rejected (i.e. the subtree under examination cannot be removed without making any difference to the tree output), the function `SetTestingToFailed` is called; if this function returns a value of false, the program is aborted due to an internal error. Alternatively, if the outcome of the permutation test is to accept the pruning proposal (i.e. the subtree under examination can be removed without affecting the output of the tree), the function `SetTestingToPruned` is called.

Note there is a subtlety here in the interpretation of the permutation test. The outcome of the test is either to reject or accept the null hypothesis that the two subjects—the data in arrays D_u and D_p—do not differ. Rejection of the null hypothesis means that the original and modified trees are (statistically) different and so the subtree under consideration *cannot* be replaced with a constant node without changing the tree's output. Similarly, acceptance of the null hypothesis means that the subtree in question may be replaced by a constant of a value equal to the average subtree output.

Returning to Figure 1, the pruning process can be illustrated in the following way. Initially, all node states are set to 'Untested' by the call to `SetAllToUntested`. First, the state of node '1' is set to 'Testing' to explore the hypothesis that the whole tree can be replaced by a single constant equal to the average of the tree output over the dataset. If, say, this hypothesis is rejected by the permutation test then the node's state is changed from 'Testing' → 'Failed'. The next hypothesis to be examined is to set state of '2' to 'Testing' by another invocation of `SetToTesting` which explores the possibility that the subtree formed by nodes '2', '3'and '4' can be replaced by a constant with the value of the average output of '2'. If the ensuing permutation test rejects the hypothesis then the state of '2' is set to 'Failed' and the procedure continues. If, on the other hand, the hypothesis that the subtree comprising nodes '2','3' and '4' can be replaced by a single, constant node, then the state of '2' is set to 'Pruned' by an invocation of the function `SetTestingToPruned`.

In practice, we examine the set of proposals to remove all possible (non-constant) nodes in the tree. In principle, we could examine a tree top-down and as soon as we identify a subtree that can be pruned, strictly there is no point in examining pruning proposals lower down in that subtree. In terms of implementation, however, the permutation testing procedure (Section 3), and in particular, its multiple comparison procedures, require that we know the total number of hypothesis tests *before* making a prune/no prune decision. Top-down pruning would thus be paradoxical. Nonetheless, the tree evaluation procedure after pruning terminates recursive evaluation soon as it encounters a 'Pruned' subtree; we therefore do take advantage of the first pruning encountered in the recursive tree traversal thereby maximising the degree of tree's simplification.

For the sake of statistical validity, conventional machine learning prescribes the use of three disjoint datasets: a training set, a validation set, and a test set—see [14, p.222], for example. The training set is used for parameter adjustment and yields the so-called *substitution error* that is usually a wildly optimistic estimate of generalisation performance, and is of little significance beyond model training. Given some number of competing trained models, the validation set is used to select one model for adoption—formally, a *model selection* stage. Performance over the validation set is generally an optimistic estimate of the chosen model's generalisation error since the model has been selected based on its performance over the (finite) validation set. Finally, an estimate of the model's generalisation performance is obtained from the test set although since this too is finite, the estimate is uncertain but hopefully unbiased. In the context of tree pruning, we are performing a *model selection* procedure: that is, given a choice between two models—the original, as-evolved tree and the pruned tree, we accept a given pruning proposal if the two model responses are 'identical' since, by definition, the pruned model is simpler and therefore to be preferred. In summary, we employ the validation set for tree pruning since this is neither a training process nor a testing process.

3 Permutation Testing

Permutation tests were originally devised by the English statistician Ronald Fisher in the 1930s as a means of illustrating hypothesis tests. Given two groups of subjects $\mathcal{A}$ and $\mathcal{B}$, both of size n, with two values of some statistic T, $T(\mathcal{A})$ and $T(\mathcal{B})$ computed over each group, respectively, and an observed difference $\delta_{\mathcal{A},\mathcal{B}} = T(\mathcal{A}) - T(\mathcal{B})$ where $\delta_{\mathcal{A},\mathcal{B}} \in \Delta_{\mathcal{A},\mathcal{B}}$. The null hypothesis H_0 assumes that groups $\mathcal{A}$ and $\mathcal{B}$ are drawn from the same population so the expectation value of $\delta_{\mathcal{A},\mathcal{B}}$, therefore, should be identically zero. If the null hypothesis is true then we are at liberty to randomly allocate the $2n$ data in $\mathcal{A} \cup \mathcal{B}$ to either one of two test groups, say, $\mathcal{C}$ and $\mathcal{D}$, allocating n data to each, and compute a new value of test statistic $\delta_{\mathcal{C},\mathcal{D}}$. If we repeat this random allocation to $\mathcal{C}$ or $\mathcal{D}$ a large number of times, each time obtaining a different value of $\delta_{\mathcal{C},\mathcal{D}}$, we can obtain a distribution of $\Delta_{\mathcal{C},\mathcal{D}}$, the so-called *permutation distribution*. Under the null hypothesis $\langle \delta_{C,D} \rangle = 0$.

By computing the probability that the actual observed difference $\delta_{\mathcal{A},\mathcal{B}}$ could have been drawn from the permutation distribution, we have three potential outcomes for the permutation test:

1. $\mathcal{A}$ and $\mathcal{B}$ are (statistically) identical and so the given pruning proposal can be accepted.
2. The performance of group $\mathcal{A}$ is better than that of $\mathcal{B}$. If $\mathcal{A}$ is from the pruned tree and we can accept the pruning proposal since improving the performance of the tree is (probably) beneficial.
3. The performance of $\mathcal{A}$ is worse than $\mathcal{B}$; if again, $\mathcal{A}$ are the pruned responses then we wish to reject that pruning proposal.

In practice, we estimate the p-value by counting the number times $\delta_{C,D}$ falls in a given range over a large number of trials.

Since we are concerned with rejecting a pruning proposal only if it produces a *worse* outcome, we adopt a one-sided hypothesis test. Permutation tests are described in greater detail in [15–17].

(We should at this point note a difference in terminology in the literature. Fisher's original thought experiment to illustrate hypothesis testing involved assembling the permutation distribution with all $n!$ exhaustive permutations of the data. For even modest values on n, of course, this exact approach becomes infeasible, and practitioners typically approximate the permutation distribution using resampling—as described above—leading to the alternative name of a *randomization test*. Here we adopt the commonly-used terminology of "permutation test" for the resampling procedure.)

In the context of GP tree pruning in a regression problem, we can consider the two groups of squared errors for each of the n data records in the validation set as the two groups $\mathcal{A}$ and $\mathcal{B}$. One is obtained from the original, as-evolved tree and the other by replacing the given subtree with a constant, as described in Section 2.

Two key technical points need to be considered at this point: firstly, whether pruning affects the generalisation ability of the model. The procedure being implemented here is *model selection*. (It is for that reason we use the validation set for the permutation test rather than the training set or a test set—generalisation error is estimated over a test set.) To select between two models—pruned and unpruned—we pose the question: does the unpruned tree exhibit the (statistically) same error over the validation set? If this question is answered in the affirmative then we prefer the the simpler tree according to Occam's razor. In reality, this question only makes any sense with respect to the validation set. If more data were available then we could use them both improve the training and to reduce the variance on the model selection decision. Consequently, questions about generalisation should be unconnected to the pruning decisions. We hypothesise that accepting a pruning proposal on the basis of comparing validation set errors will have no systematic effect on the test (generalisation) error. In essence, if we have an (obvious) redundancy of the form, say, $y = f(x) + x - x$ then simplifying this to $y = f(x)$ will not affect the generalisation error of the model. Thus we conjecture that pruning will sometimes reduce test error and sometimes increase it, but overall, will have no statistically significant effect; this point is addressed further in Section 5.1.2.

In terms of implementation, the quantities we are permuting are the two groups of n squared residuals obtained with and without a pruning proposal. Consequently, it is only necessary to calculate these once at the start of a subtree pruning process and cache them. The computational demands of a given permutation test are thus modest. Further, we are interested in the decision about whether pruning some given subtree results in (statistically) the same tree semantics as the *original, as-evolved* tree, and not the last pruned tree. Consequently, our 'reference' responses from the as-evolved, unpruned tree do not change during the sequence of pruning decisions for the whole tree.

The second issue concerns multiple comparison procedures [18]. In this work, we are computing multiple test statistics, and there is a well-known issue with such comparisons tending to increase the error rate over families of tests [18]. (We think it self-evident that the set of tests we employ does comprise a *family* [18].) The concept of MCPs is well-known to the GP community albeit in the guise of the post hoc corrections that usually follow Friedman tests of group homogeneity [19]. In hypothesis testing there are two sorts of error: Type I error, which occurs with probability α, is the situation where a null hypothesis is erroneously rejected. Complementary to this, a Type II error, which occurs with probability β, is the mistaken rejection of the alternative hypothesis; the quantity

$(1-\beta)$ is denoted the *power* of the test. When α increases, β decreases, and vice versa although the exact functional form is hard to specify in general. In the context of the present work, a Type I error will lead to the rejection of a (perfectly good) pruning proposal and is benign (except that a possible opportunity to simplify the tree has been missed). A Type II error, on the other hand, leads to accepting a pruning proposal that should really have been rejected. Consequently, controlling Type II error is of greater importance in this work. Most multiple comparison procedures (MCPs), however, have been focussed on controlling Type I error although the *false discovery rate* (FDR) procedure of Benjamini and Hochberg [20] has been shown to maintain test power and is therefore appropriate here. In the following sections of experimental results, we consider both the use and omission of MCPs.

4 Experimental Methodology

We have employed a fairly standard generational GP search in this work [21]. The parameters of the algorithm are summarised in Table 1. We have used 10% elitism and therefore generated the remainder of the child population by always applying crossover and mutation to ensure population diversity. If a breeding operation generated a tree larger than the pre-specified hard node count limit, one of the parent trees was randomly selected for copying into the child population.

Table 1: Parameters used for generational GP evolution.

Population size	100
Population initialisation	Random tree sizes between one and node count limit
Elitism	10%
No. of tree evaluations	20,000
Internal nodes	Add, subtract, multiplication, analytic quotient [22]
Terminal nodes	Independent variable, constant $\in \{0.1, 0.2, \ldots, 0.9, 1.0\}$
Crossover	Point crossover with 90% probability of selecting internal node [21]
Crossover probability	1.0
Mutation	Point mutation [21]
Mutation probability	1.0
Mutation tree depth	4
Hard node count limit	$[20 \ldots 100]$

We have employed a set of ten univariate regression test functions that have previously been used in the GP literature—see Table 2. The first four of these functions are, in principle, representable exactly with the internal nodes used in the GP. The remaining functions require appropriate approximation. We have used training sets of 20 data randomly sampled over the domain, and validation sets of the same size; the test sets comprised 1,000 data in order to obtain reliable estimates of generalisation error. The rather small size of 20 training/validation data was quite deliberate and is related to the challenge presented by various test functions for GP, which has received some attention [23]. In particular, the difficulty associated with learning high-dimensional problems stems from the exponentially-decreasing density of data with increasing problem dimensionality—the so-called 'curse of dimensionality'. At the same time, we wanted to focus on univariate problems to facilitate more straightforward analysis of the prune subtrees (although, in the event, this proved optimistic). Hence the choice of 20 training data so as to pose set of fairly ill-conditioned and therefore challenging learning problems; a 50:50 split between training and validation sets is commonplace in machine learning practice.

We have standardised on the range of node limits $[16 \ldots 256]$ which corresponds, under the assumption of all binary internal nodes, to tree depths of $[4 \ldots 8]$, a range typically employed to solve problems such as the test functions used here.

Table 2: Test functions used in this work.

Function	**Equation**	**Domain**
Cubic polynomial [24]	$y = x^3 + x^2 + x$	$[-1 \ldots +1]$
Quartic polynomial [24]	$y = x^4 + x^3 + x^2 + x$	$[-1 \ldots +1]$
Quintic polynomial [24]	$y = x^5 + x^4 + x^3 + x^2 + x$	$[-1 \ldots +1]$
Sextic polynomial [24]	$y = x^6 + x^5 + x^4 + x^3 + x^2 + x$	$[-1 \ldots +1]$
French curve [25]	$y = 4.26 \exp^{-x} - 4 \exp^{-2x} + 3 \exp^{-3x}$	$[0 \ldots 3.25]$
Uy_5 [24]	$y = \sin(x^2) \times \cos(x) + 1$	$[-1 \ldots +1]$
Uy_6 [24]	$y = \sin(x) + \sin(x + x^2)$	$[-1 \ldots +1]$
Uy_7 [24]	$y = \log(x + 1) + \log(1 + x^2)$	$[0 \ldots 2]$
Uy_8 [24]	$y = \sqrt{x}$	$[0 \ldots 4]$
Salustowicz [26]	$y = x^3 \exp^{-x} \cos(x) \sin(x)[\sin^2(x) \cos(x) - 1]$	$[0 \ldots 10]$

Throughout this work, we have use the commonly-accepted significance level of 5% for permutation testing. In other words, a 1-in-20 chance that the observed statistic could have been obtained fortuitously. We have used 10,000 samples for the permutation testing, a figure arrived at simply by increasing the number of trials from a low starting point until the estimators stabilised. (Ultimately, a procedure due to Gandy [27] could be implemented to minimise the number of permutation test samples, if deemed necessary.)

5 Results

After evolving a final population of GP regression trees, we selected the individual with the smallest training error for subsequent pruning. We have applied our permutation-test-based pruning procedure to establish the influence of these statistical modifications. Further, we have explored the effect of pruning on the generalisation errors in Sections 5.1.2.

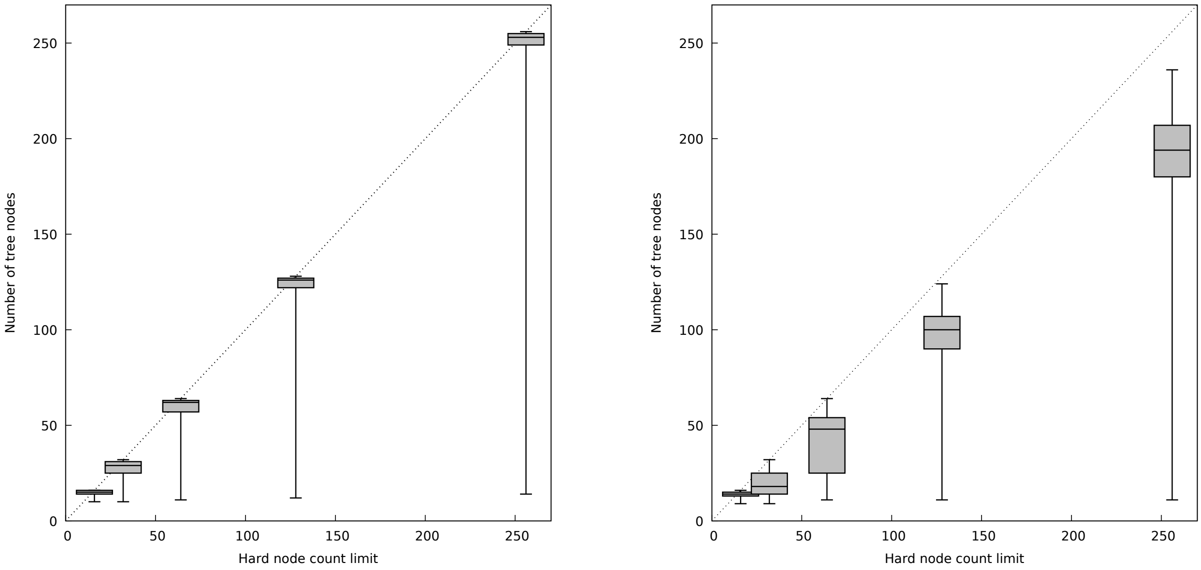
5.1 Pruned Tree Sizes

The distributions of tree node counts for all ten test functions for a range of hard node count limits are shown in Figures 2-11; these have been accumulated over 1,000 repetitions, each with a different initial population. (All results in this subsection have incorporated the Benjamini and Hochberg multiple comparison procedures.)

Figures 2(a)-11(a) show the node counts for the final evolved individuals with the smallest training errors *before* pruning. We only compare distributions over trees for which subsequent pruning was possible, and ignored (the small minority of) trees were no pruning occurred.

It is clear from Figures 2(a)-11(a) that for a given hard node count, the GP evolution tended to produce trees almost exactly this size and with small interquartile ranges although some smaller trees were generated as evidenced by the lower whiskers on the box plots. In the cases of the cubic polynomial there is some evidence of a lower bound on the evolved tree sizes of about 20 nodes. For the remaining functions, the lower bound on tree size appears to increase with hard node count limit.

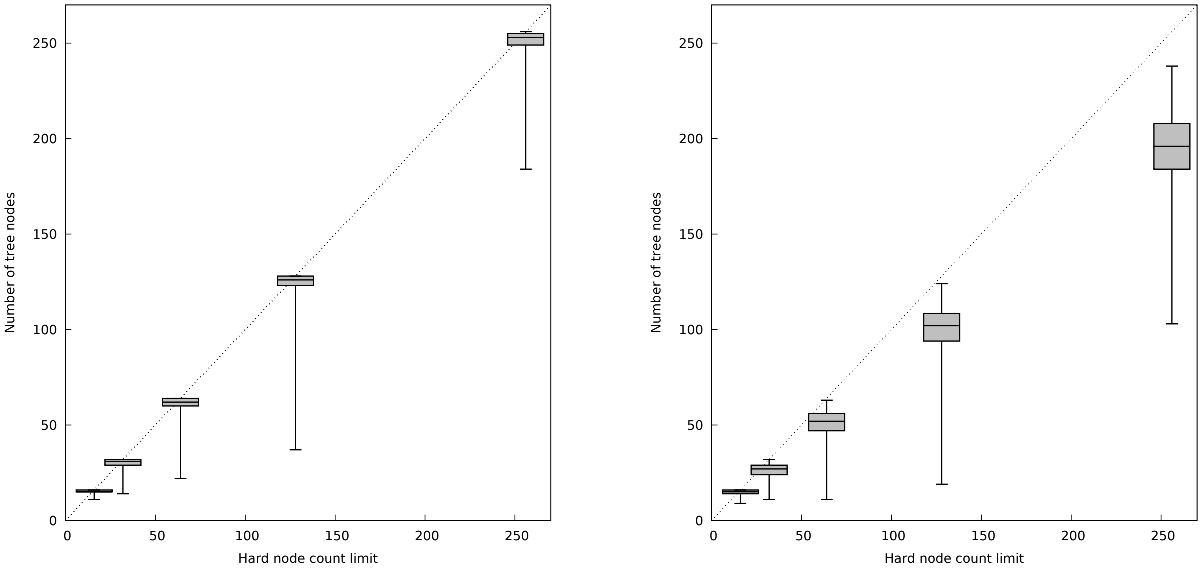
The node count distributions after pruning are shown in Figures 2(b)-11(b). There appears to be a roughly 20% consistent reduction in median tree size independent of test function although comparing with the corresponding boxplots before pruning, there is a noticeable increase in the interquartile ranges. That is, more variability in the range of tree sizes after pruning. From the upper whiskers of the box plots, it is clear that in an appreciable number of cases the trees have been reduced in size only marginally. The lower whiskers display similar characteristics as the distributions before pruning. One notable exception is the Salustowicz function (Figure 11(b)) where the lower bound on pruned tree size is unity. In other words, a number of these trees have been pruned to a single node for this test function independent of initial hard node limit.



(a) No pruning

(b) Pruned

Figure 2: Distributions of tree node counts vs. hard node count limit for the cubic polynomial test function; with MCP.

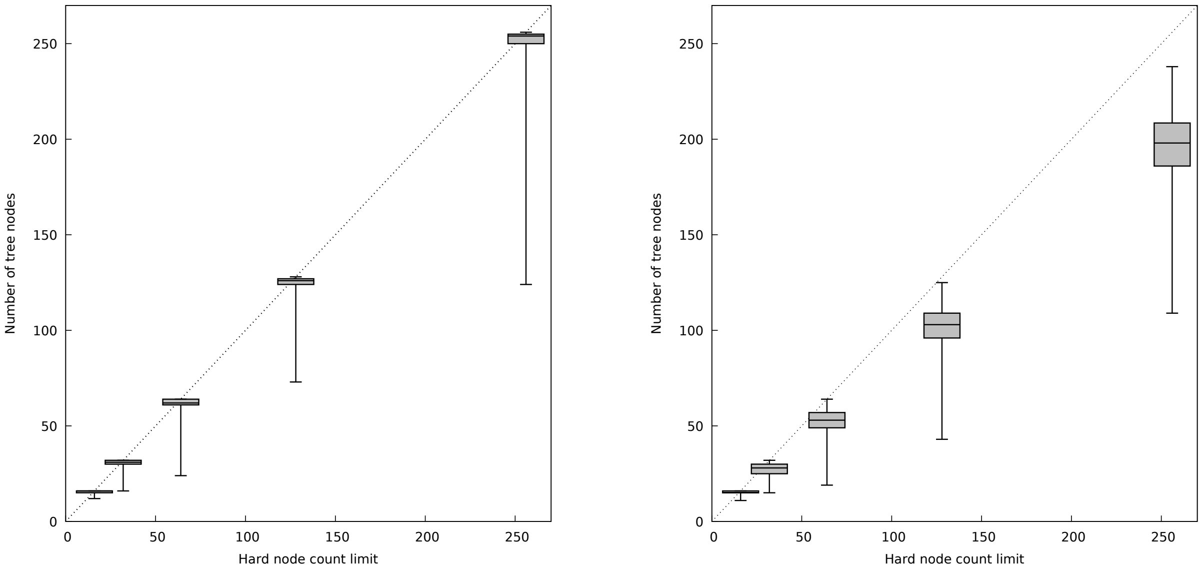


(a) No pruning

(b) Pruned

Figure 3: Distributions of tree node counts vs. hard node count limit for the quartic polynomial test function; with MCP.

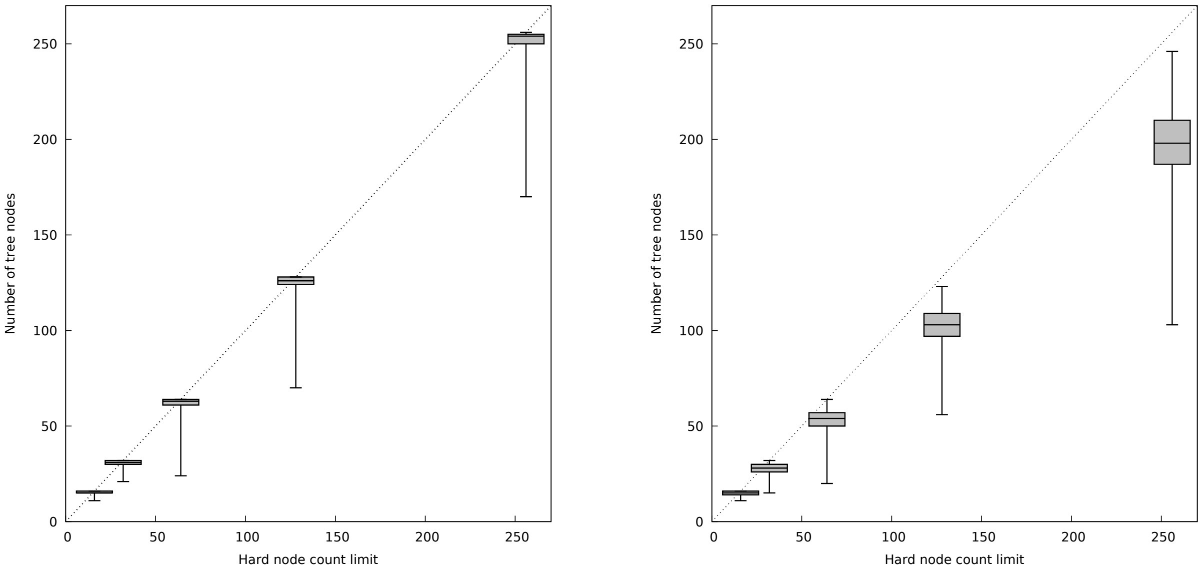
Figure 12 shows the typical relationships between unpruned and pruned individuals for the French curve test function and a hard node count limit of 128. Points on the two parallel axes show the unpruned tree sizes (left) and pruned tree sizes (right) for the same individuals before and after pruning. For comparison, the two distributions of individuals are shown as boxplots for unpruned (left) and pruned (right) populations. It is clear from this figure that unpruned individuals with an



(a) No pruning

(b) Pruned

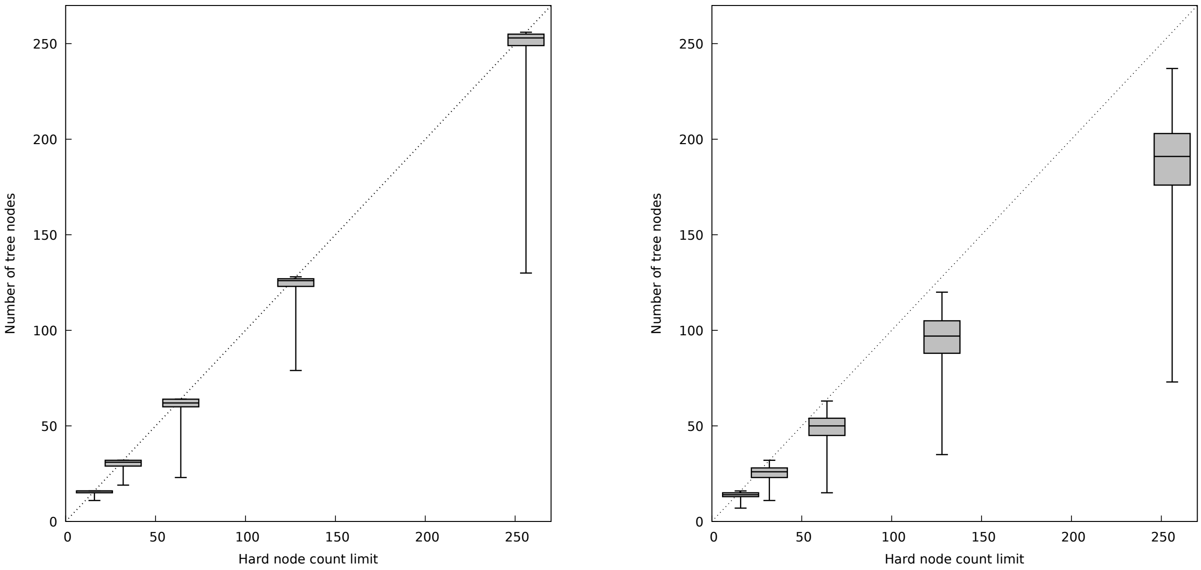
Figure 4: Distributions of tree node counts vs. hard node count limit for the quintic polynomial test function; with MCP.



(a) No pruning

(b) Pruned

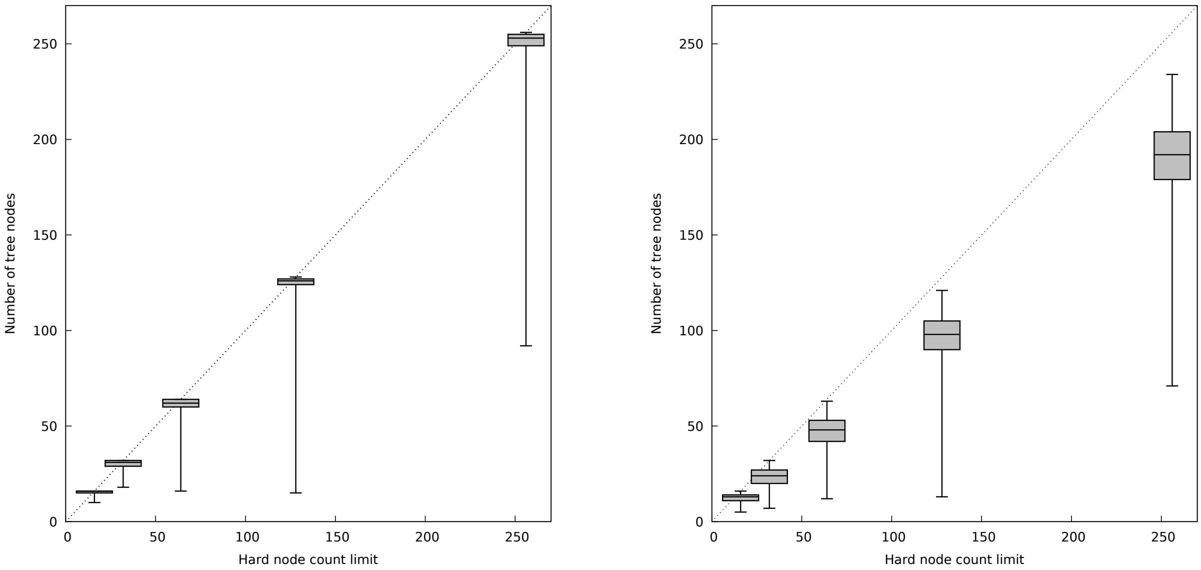
Figure 5: Distributions of tree node counts vs. hard node count limit for the sextic polynomial test function; with MCP.



(a) No pruning

(b) Pruned

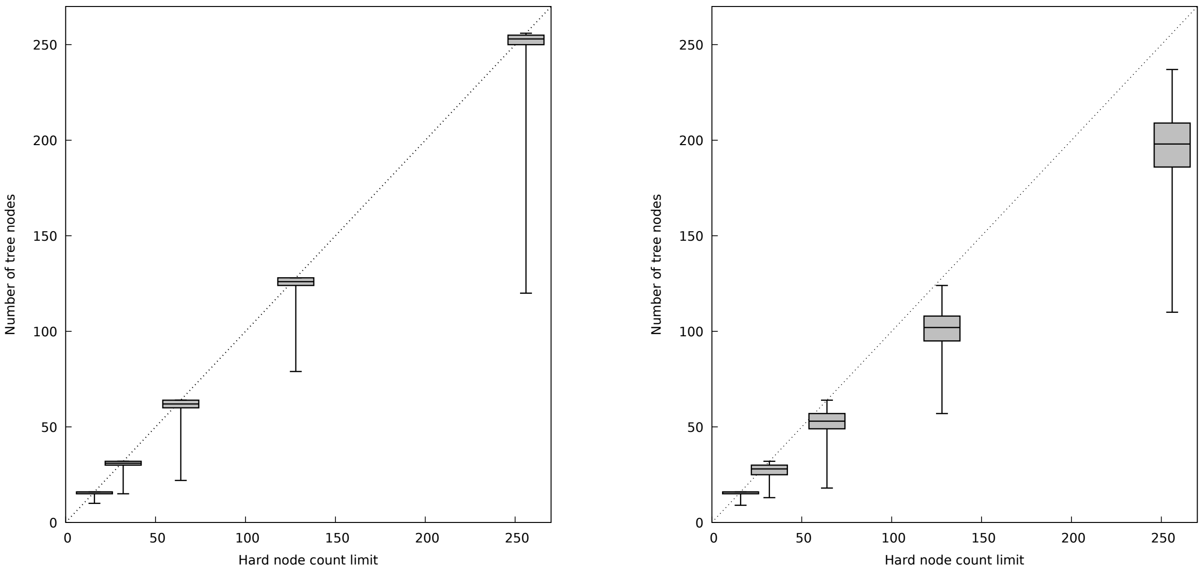
Figure 6: Distributions of tree node counts vs. hard node count limit for the French curve test function; with MCP.



(a) No pruning

(b) Pruned

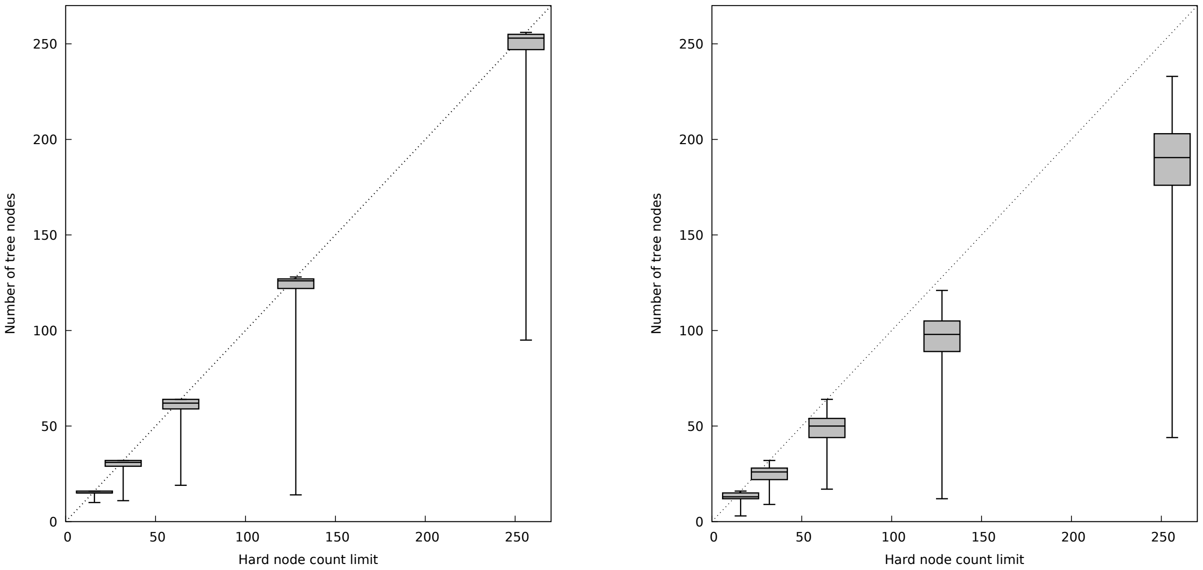
Figure 7: Distributions of tree node counts vs. hard node count limit for the Uy_5 test function; with MCP.



(a) No pruning

(b) Pruned

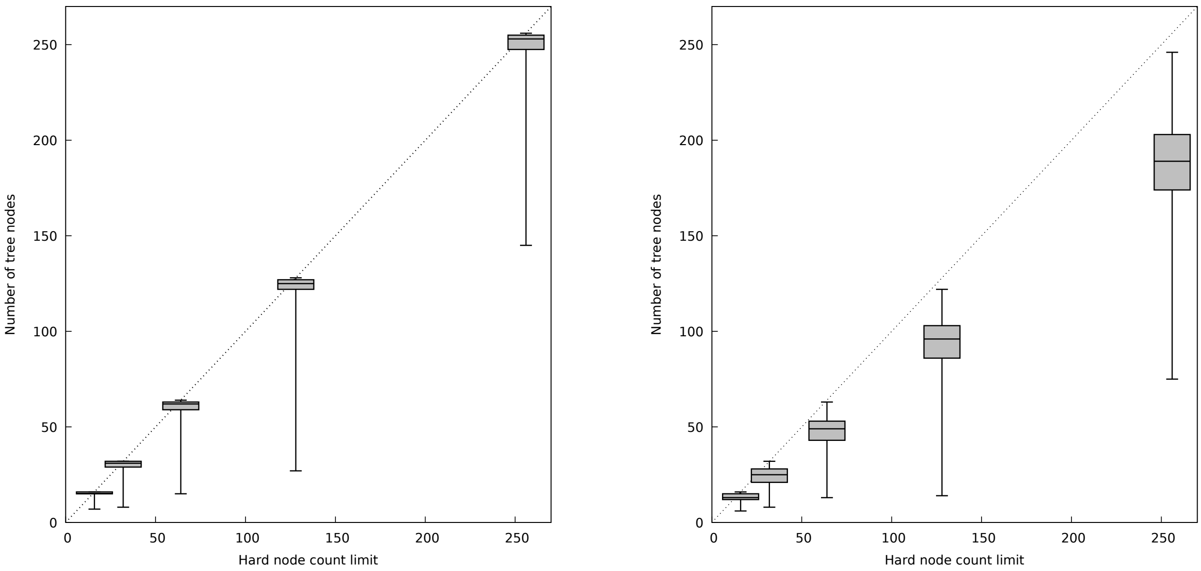
Figure 8: Distributions of tree node counts vs. hard node count limit for the Uy_6 test function; with MCP.



(a) No pruning

(b) Pruned

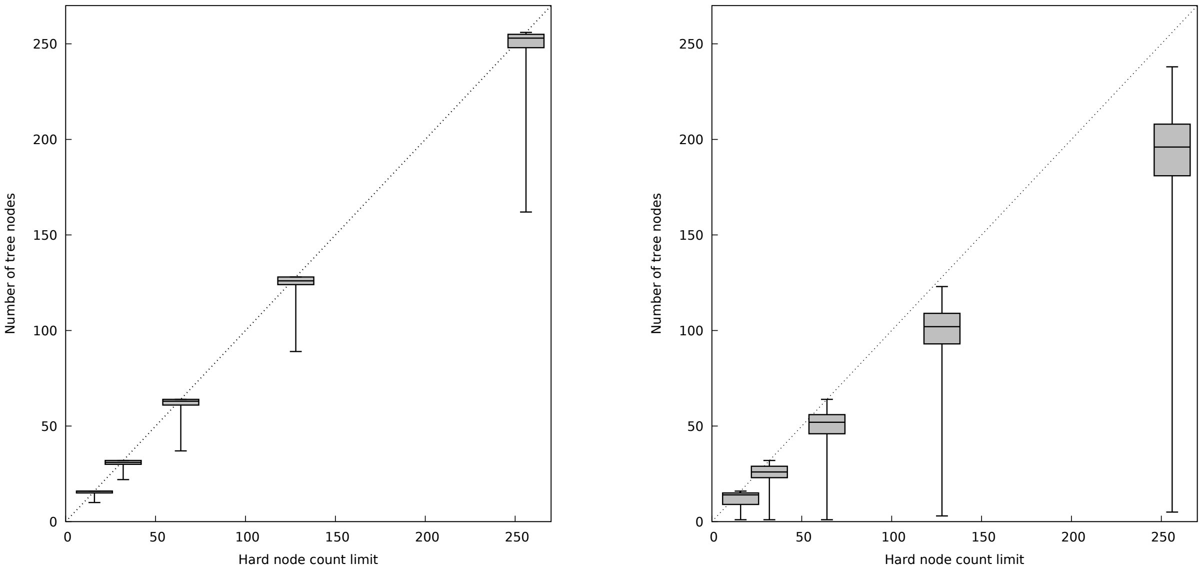
Figure 9: Distributions of tree node counts vs. hard node count limit for the Uy_7 test function; with MCP.



(a) No pruning

(b) Pruned

Figure 10: Distributions of tree node counts vs. hard node count limit for the Uy_8 test function; with MCP.



(a) No pruning

(b) Pruned

Figure 11: Distributions of tree node counts vs. hard node count limit for the Salustowicz test function; with MCP.

interquartile range of [123 . . . 127] are being pruned to a wide range of tree sizes with a corresponding interquartile range of [88 . . . 105]. The median size is being reduced from 126 to 97. Although some small unpruned individuals are being reduced by only modest extents, there are also large individuals (∼110-128 nodes) being pruned down to between 60 and 80 nodes. A few are being pruned to ∼40 nodes.

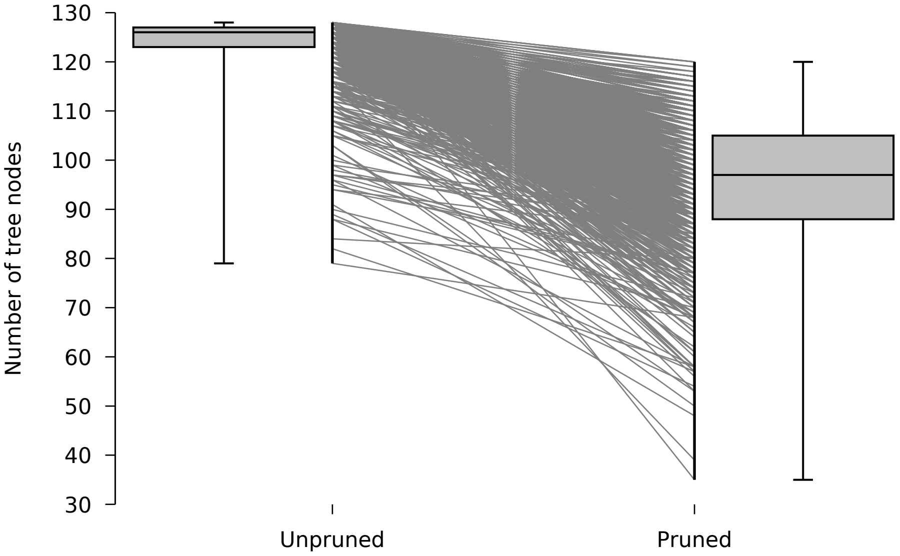


Figure 12: Parallel axis plot for the French curve test function and hard node limit of 128 showing the co-relations between unpruned and pruned individuals. The boxplots show the distributions of tree sizes before and after pruning for 1,000 independently-initialised runs.

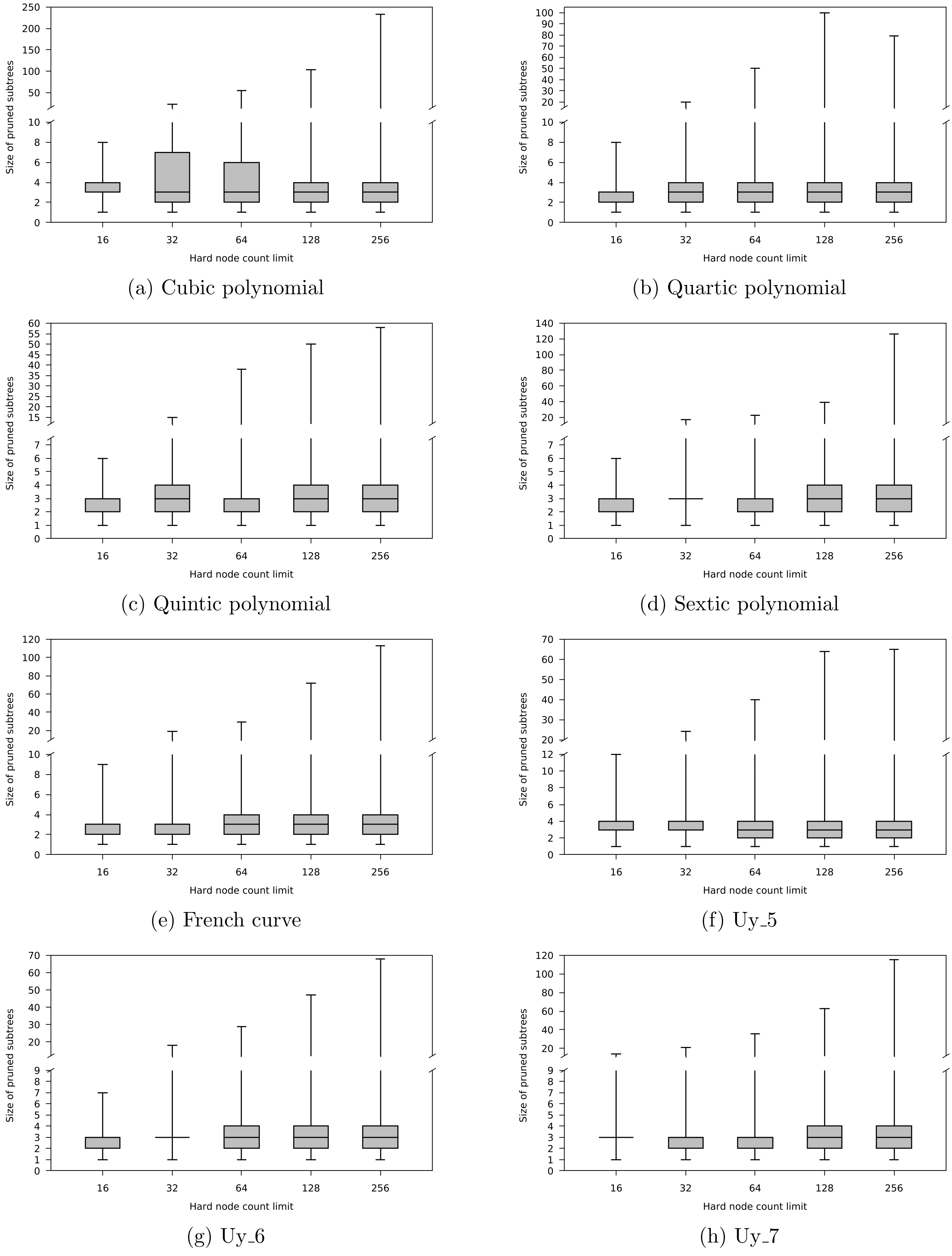
5.1.1 Distributions of Pruned Tree Sizes

The results in the previous section are for trees to which all possible pruning proposals have been applied, and are therefore reduced to their minimum size. We have paid close attention, however, to the *individual* accepted pruning proposals. In particular, Figure 13 shows the distributions of the sizes of pruned subtrees for each of the test functions. A noteworthy feature of these distributions is their small interquartile ranges. Most of the pruned trees tend to be rather small in size although a few trees up to around several hundred nodes are also removed by pruning. The other noteworthy feature of Figure 13 is that the median size of pruned trees is around three nodes with comparatively little variation with test function/hard node count limit. It thus seems highly likely that the most commonly pruned tree comprises a binary node and either two terminals or two constants.

We have examined the distributions of pruned tree sizes in Figure 13 in greater detail paying particular attention to trees of three nodes, and this analysis is shown in Tables 3-12.

Table 3: Distribution of pruned 3-node subtrees: Cubic polynomial

Node limit	16	32	64	128	256
# pruning events	1,566	5,908	10,138	17,300	36,558
% 3-node trees	51.15	37.39	38.41	40.21	38.68
% const $\circledast$ const	64.79	40.83	54.65	64.62	65.94
% $(x - x)$	25.72	28.56	19.34	17.42	17.04
% 3-node algebraic	90.51	69.4	73.99	82.04	82.99
% other binary	4.37	20.28	16.64	9.95	8.85
% 3-node unary	5.12	2.31	3.16	4.26	4.76



(a) Cubic polynomial

(b) Quartic polynomial

(c) Quintic polynomial

(d) Sextic polynomial

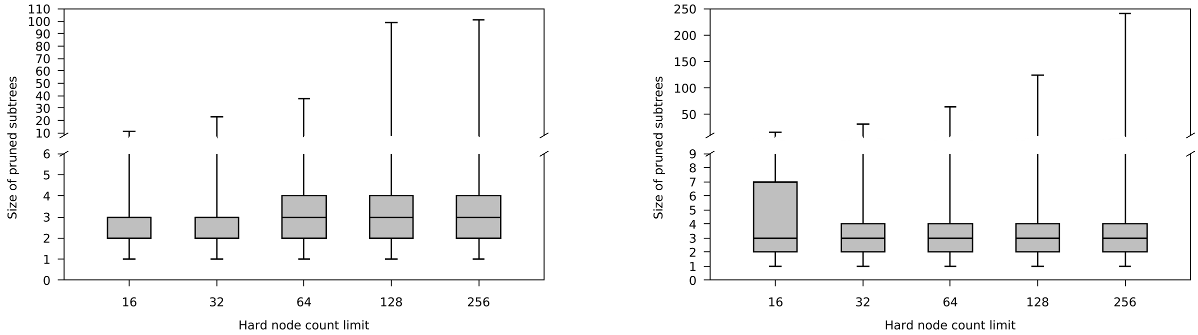
(e) French curve

(f) Uy_5

(g) Uy_6

(h) Uy_7

Figure 13: Distributions of the sizes of pruned subtrees.



(i) Uy_8

(j) Salustowicz

Figure 13 (cont'd): Distributions of the sizes of pruned subtrees.

Table 4: Distribution of pruned 3-node subtrees: Quartic polynomial

Node limit	16	32	64	128	256
# pruning events	1,301	3,147	6,594	14,634	34,326
% 3-node trees	49.81	45.03	44.24	43.06	39.35
% const ⊛ const	83.95	65.63	68.02	70.47	68.21
% $(x - x)$	5.25	20.75	17.79	18.03	17.39
% 3-node algebraic	89.20	86.38	85.81	88.50	85.60
% other binary	0.46	6.99	7.13	4.93	6.67
% 3-node unary	10.19	4.73	4.35	4.65	5.36

Table 5: Distribution of pruned 3-node subtrees: Quintic polynomial

Node limit	16	32	64	128	256
# pruning events	1,300	2,434	5,594	13,822	33,921
% 3-node trees	41.00	48.77	47.50	43.42	39.06
% const ⊛ const	78.80	73.13	73.54	71.67	67.58
% $(x - x)$	4.13	17.35	17.20	17.85	17.77
% 3-node algebraic	82.93	90.48	90.74	89.52	85.35
% other binary	0.94	3.20	3.20	3.78	6.77
% 3-node unary	16.14	5.64	4.82	5.05	5.28

Table 6: Distribution of pruned 3-node subtrees: Sextic polynomial

Node limit	16	32	64	128	256
# pruning events	1,250	2,257	5,283	13,681	33,267
% 3-node trees	42.16	50.16	48.78	42.86	39.65
% const ⊛ const	73.81	74.12	74.89	70.28	68.72
% $(x - x)$	12.14	18.64	16.72	18.55	17.15
% 3-node algebraic	85.96	92.76	91.62	88.83	85.87
% other binary	0.00	1.33	2.52	4.47	6.53
% 3-node unary	14.04	5.57	5.20	5.05	5.31

Table 7: Distribution of pruned 3-node subtrees: French curve

Node limit	16	32	64	128	256
# pruning events	1,411	3,275	7,494	17,353	37,845
% 3-node trees	50.32	47.11	44.88	41.87	38.56
% const $\circledast$ const	91.27	80.62	73.48	69.29	66.44
% $(x - x)$	2.68	10.89	14.36	15.14	15.95
% 3-node algebraic	93.94	91.51	87.84	84.43	82.39
% other binary	0.14	2.53	5.08	7.85	9.15
% 3-node unary	5.77	4.93	5.32	4.78	5.13

Table 8: Distribution of pruned 3-node subtrees: Uy_5

Node limit	16	32	64	128	256
# pruning events	2,168	4,216	8,469	16,708	36,552
% 3-node trees	47.05	44.17	42.02	41.24	38.86
% const $\circledast$ const	87.25	77.12	71.45	70.42	68.03
% $(x - x)$	1.86	13.16	17.25	16.75	17.47
% 3-node algebraic	89.12	90.28	88.70	87.17	85.51
% other binary	0.88	2.26	4.33	5.09	6.97
% 3-node unary	10.00	6.50	5.59	5.92	5.09

Table 9: Distribution of pruned 3-node subtrees: Uy_6

Node limit	16	32	64	128	256
# pruning events	1,231	2,419	5,596	14,264	32,937
% 3-node trees	50.77	52.75	47.89	45.34	41.48
% const $\circledast$ const	87.20	75.08	71.57	71.42	69.09
% $(x - x)$	6.88	17.55	19.63	17.84	17.18
% 3-node algebraic	94.08	92.63	91.19	89.27	86.27
% other binary	0.00	1.88	3.25	4.50	6.85
% 3-node unary	5.92	4.78	4.59	4.59	4.27

Table 10: Distribution of pruned 3-node subtrees: Uy_7

Node limit	16	32	64	128	256
# pruning events	1,813	3,170	7,081	16,125	36,824
% 3-node trees	55.49	49.87	47.31	43.14	41.00
% const $\circledast$ const	86.88	78.05	74.75	70.99	67.01
% $(x - x)$	5.47	13.35	15.55	16.89	17.03
% 3-node algebraic	92.35	91.40	90.30	87.88	84.04
% other binary	1.29	2.78	3.64	5.94	8.58
% 3-node unary	6.26	4.93	4.36	3.87	4.46

Table 11: Distribution of pruned 3-node subtrees: Uy_8

Node limit	16	32	64	128	256
# pruning events	1,684	3,512	7,515	17,573	37,678
% 3-node trees	55.29	49.97	45.56	42.76	40.45
% const $\circledast$ const	87.22	80.46	74.59	68.90	67.85
% $(x - x)$	5.59	12.02	14.52	16.28	16.06
% 3-node algebraic	92.80	92.48	89.11	85.17	83.90
% other binary	0.64	2.28	4.99	7.21	8.71
% 3-node unary	6.44	4.62	4.35	5.04	4.51

Table 12: Distribution of pruned 3-node subtrees: Salustowicz

Node limit	16	32	64	128	256
# pruning events	3,080	3,862	6,673	14,202	32,942
% 3-node trees	30.29	37.36	45.62	45.54	42.86
% const $\circledast$ const	65.92	69.58	74.31	72.89	70.27
% $(x - x)$	0.54	9.08	12.55	14.21	15.85
% 3-node algebraic	66.45	78.66	86.86	87.10	86.13
% other binary	15.65	8.45	6.93	6.28	6.87
% 3-node unary	7.72	6.72	4.17	4.31	4.40

It is clear that the number of successful prunings increases with increasing hard node count limit. This is not surprising as larger trees give greater scope for generating redundant, and therefore prunable, genetic material. The percentage of these pruning events that were median-sized 3-node trees (indicated in the labels as "% 3-node trees") is around 40-50% for node count limits of 16 and falls to 35-40% for a limit of 256 except for the Salustowicz function which exhibits slightly lower figures. (The reason is not clear but the results from the Salustowicz function often stand apart in this report, probably due to the challenge of learning a function with so many extrema with just 20 data.)

Of the pruned 3-node trees summarised in Tables 3-12, around 65-90% are binary nodes having two constant children ("% const $\circledast$ const, where $\circledast$ denotes any of the binary operations used) with a smaller proportions of the form $x - x$. It is thus clear that the permutation-based pruning approach presented here is successfully performing algebraic simplification as well as approximate simplification.

Tables 3-12 also show the percentages of other 3-node, binary-rooted subtrees of the form '$x \circledast$const' (or 'const$\circledast x$') labelled "% other binary"; removal of a small number of these is equivalent to applying a simplification[1] of the form '$x \times 1 \rightarrow x$'. Further, some percentage of pruned 3-node subtrees are rooted with a unary node which appear mostly to be algebraic simplifications of the form '$-(-$const)', that is, two consecutive, cancelling unary minus operations.

From the above results, it thus seems clear that at least ~80-95% of the 3-node prunings represent algebraic simplifications being carried out—by definition—at the peripheries of trees. These 3-node simplifications, however, account for only around half of the total number of pruning events. Some of the pruned trees larger than three nodes may well be producing algebraic simplification, but this is difficult to verify due to the increasing number of ways an equivalent expression may be written. Some more insight, however, may be gained by examining what fraction of the permutation tests returned a precisely zero probability of rejecting the null hypothesis. Such a result can only be delivered when the two trees' responses—with vs. without pruning—are absolutely identical implying that the pruning being considered represents an algebraic simplification. These percentages are shown in Table 13

[1] In the GP framework used here, constants cannot be equal to zero so simplifications of the form '$x \pm 0 \rightarrow x$' and '$x \times 0 \rightarrow 0$' are not possible.

for each test function and hard node limit, and for 3-node trees as well as pruned trees larger than three nodes. It is clear that, in general, at least the high nineties of percent of pruning probabilities are zero implying that these represent algebraic simplification. A slightly smaller percentage of trees larger than three nodes return zero probability than the corresponding figure for 3-node trees although this is reasonable given that there is greater scope for larger trees to be approximately rather than algebraically equivalent.

Table 13: Percentages of zero probability values return by the permutation tests for pruned subtrees $= 3$ nodes, and > 3 nodes, by hard node count limit, and for different test functions.

Node limit	16		32		64		128		256	
	$= 3$	> 3	$= 3$	> 3	$= 3$	> 3	$= 3$	> 3	$= 3$	> 3
Cubic	97.38	96.08	99.59	98.86	99.44	98.65	99.11	97.90	98.44	96.63
Quartic	99.07	99.08	98.94	98.03	99.14	97.88	99.00	97.82	98.45	96.66
Quintic	97.19	98.04	99.58	98.72	99.17	97.68	99.33	98.01	98.13	96.19
Sextic	99.81	99.86	99.73	98.67	99.26	98.37	99.06	97.70	98.45	96.62
French curve	98.59	98.43	98.12	95.03	96.13	90.87	95.31	88.97	95.33	89.78
Uy_5	98.92	98.17	99.46	98.90	99.44	98.70	98.72	97.20	98.11	96.21
Uy_6	99.68	100.00	99.76	99.56	99.29	98.63	99.26	98.05	98.40	96.37
Uy_7	98.31	96.65	97.98	94.21	98.54	95.58	98.66	96.15	97.91	95.13
Uy_8	98.82	98.80	98.18	96.19	97.98	94.21	97.22	93.47	96.77	92.07
Salustowicz	77.06	52.03	86.56	64.94	92.81	75.34	95.65	85.53	96.27	90.02

5.1.2 Effect of Pruning on the Generalisation Error

If pruning makes no difference to the model selection decision (i.e. validation set error) other than producing a smaller tree, we have conjectured that pruning can have no effect on the *average generalisation error* of the tree. We have explored this hypothesis by generating a (third) test set [14, p.222] of 1,000 data independent of both the training and validation sets to estimate the generalisation errors of 1,000 trees generated from independent GP runs both before and after pruning; we removed from this experiment trees that have remained unpruned.

If pruning does not affect generalisation performance—which is, by definition, the average error over test sets— then by chance we would expect exactly half the (pruned) trees to exhibit an improved test error after pruning and half to have a higher test error after pruning. This null hypothesis, therefore, has a binomial distribution with a probability $p = 0.5$, which, for 1,000 samples, is well-approximated by a normal distribution. The mean is 500 and the corresponding 95% ($1.96\,\sigma$) confidence interval is $[469 \ldots 531]$.

We have explored the situations where the null hypothesis has to be rejected with a series of one-sided hypothesis tests probing the notions that pruning (on average) either improves or degrades the generalisation error. The results are summarised in Table 14. For each of the 1,000 pruned trees examined, the generalisation error was either unchanged, increased by pruning or decreased by it. Between 58.5% and 100% of the generalisation errors were unchanged by pruning and we followed the usual statistical practice of dividing these equally between the counts for improved and degraded test errors; we explored the one-sided hypothesis for whichever count (improvement/degradation) was larger. Which test has been applied is shown in the table with either a down-arrow ($\downarrow$) for a reduction in test error, or an up-arrow ($\uparrow$) for an increase. From Table 14, it is clear that most of the null hypotheses need to be accepted (= 'no change' in test error) although there is no clear pattern concerning the rejections. Sometimes these are statistically-significant improvements in test error (at the 5% confidence level) and sometimes degradations. The Salustowicz function includes both degradations and improvements. This picture of uncertain effects on test error has also been reported in [2, 6].

Table 14: One-sided hypothesis tests for 20 training data and 20 validation data. For each hard node limit value, Z-value shows the value of the Z statistic, p the p-value of the test, and the third columns shows whether the one-sided test is for a degradation (↑) or an improvement(↓) in the generalisation error. Statistically-significant tests at the 5% confidence level are shown in bold face.

Node limit	16			32			64			128			256		
	Z-value	p		Z-value	p		Z-value	p		Z-value	p		Z-value	p	
Cubic	1.2649	0.0736	↓	0.6641	0.3477	↓	0.6641	0.3477	↓	0.6008	0.3955	↓	0.1897	0.7884	↑
Quartic	0.1265	0.8580	↓	0.6957	0.3252	↓	0.9171	0.1947	↓	0.2530	0.7205	↑	0.0949	0.8933	↑
Quintic	0.1897	0.7884	↓	0.3162	0.6547	↓	0.4427	0.5312	↓	0.0949	0.8933	↑	0.9171	0.1947	↑
Sextic	0.0000	1.0000	↓	0.2846	0.6873	↓	0.5376	0.4471	↓	0.5692	0.4208	↓	0.7273	0.3037	↓
French	0.0949	0.8933	↓	0.0316	0.9643	↓	0.5376	0.4471	↑	0.2846	0.6873	↑	**1.5495**	**0.0284**	↑
Uy_5	0.3162	0.6547	↓	0.4743	0.5023	↓	0.3162	0.6547	↓	0.0949	0.8933	↓	**2.5298**	**0.0003**	↑
Uy_6	0.0000	1.0000	↓	0.1265	0.8580	↓	0.4111	0.5610	↓	0.6008	0.3955	↓	0.4427	0.5312	↑
Uy_7	0.4743	0.5023	↓	0.6957	0.3252	↓	1.2649	0.0736	↓	**2.1820**	**0.0020**	↓	**2.8144**	**0.0001**	↓
Uy_8	0.1897	0.7884	↓	0.8222	0.2449	↓	1.2649	0.0736	↓	1.5811	**0.0253**	↓	**2.2452**	**0.0015**	↓
Salustowicz	**8.7911**	**<0.0001**	↑	**3.9212**	**<0.0001**	↑	**4.3323**	**<0.0001**	↑	**1.5495**	**0.0284**	↑	**1.9922**	**0.0048**	↓

To try to further understand the effects of pruning on generalisation, we have repeated the hypothesis tests in Table 14 for 100 training data but retaining a validation set size (that was used for pruning) of 20 data. The trees in this series of experiments should thus be better trained before pruning. The results of this second series of experiments are shown in Table 15.

A rather more consistent pattern emerges from Table 15 where all hypothesis tests are against improvements in test error. That is, tree prunings always yielded net reductions in test error. This can be understood since improved training means that the GP function better approximates the target function; pruning here is always referenced to the performance of the as-evolved tree and if this is deficiently trained then pruning will find simpler approximations to this deficient approximation. It is also apparent from this table that statistically-significant reductions tend to be for larger tree node limits. This is sensible since larger node limits tend to produce larger trees (see Figures 2(a)-11(a)) which are more likely to contain redundant subtrees.

5.1.3 Influence of the Multiple Comparison Procedure

In order to gain quantitative insight into the influence of employing the multiple comparison procedure, we have examined the distributions of subtrees that would be accepted for pruning *without* MCP but were rejected *with* MCP. In other words, the sizes of subtrees that MCP causes to be retained and not pruned. These distributions are shown in Figure 14 from which it is clear that although some large subtrees were kept due to MCP, the additional retained subtrees mostly comprise fewer than ten nodes. This figure, however, needs to be set alongside Figure 13, the size distributions of subtrees that were successfully pruned. It seems clear, therefore, that MCP is predominantly suppressing the removal of the larger candidate subtrees than is the norm without MCP.

5.1.4 CPU Timings

A typical example of the CPU timings required for the evolution and pruning operations is shown in Figure 15. The upper, solid curve shows the mean timings to complete only the evolution phase; note that as the hard node count limit reduces below ∼30, the required CPU time increases. The exact reason for this increase is not clear. Certainly for smaller node count limits, there is a greater likelihood of an offspring tree exceeding that limit, but in the GP setup used here, this results in one of the parents being randomly chosen for inclusion in the child population rather than a time-consuming, complete re-breeding/re-evaluation phase.

The lower, dotted curve shows the timings to perform a pruning procedure on a single, best-trained individual. Note that error bars are also plotted for the lower curve but are not discernible at this scale. We can make two observations from the lower curve in Figure 15: Firstly, the rate of increase of the mean timing is linear in the hard node count limit. From Figures 2(a)-11(a), it is apparent that the most likely tree size, judged by the median, is almost exactly equal to the node count limit itself. Hence the mean pruning time increases linearly with the node count limit.

The second observation from Figure 15 is that pruning produces only a modest increase in runtime over that required for evolutionary search alone—typically ∼25% with an *decreasing* time burden of pruning with increasing evolved tree size.

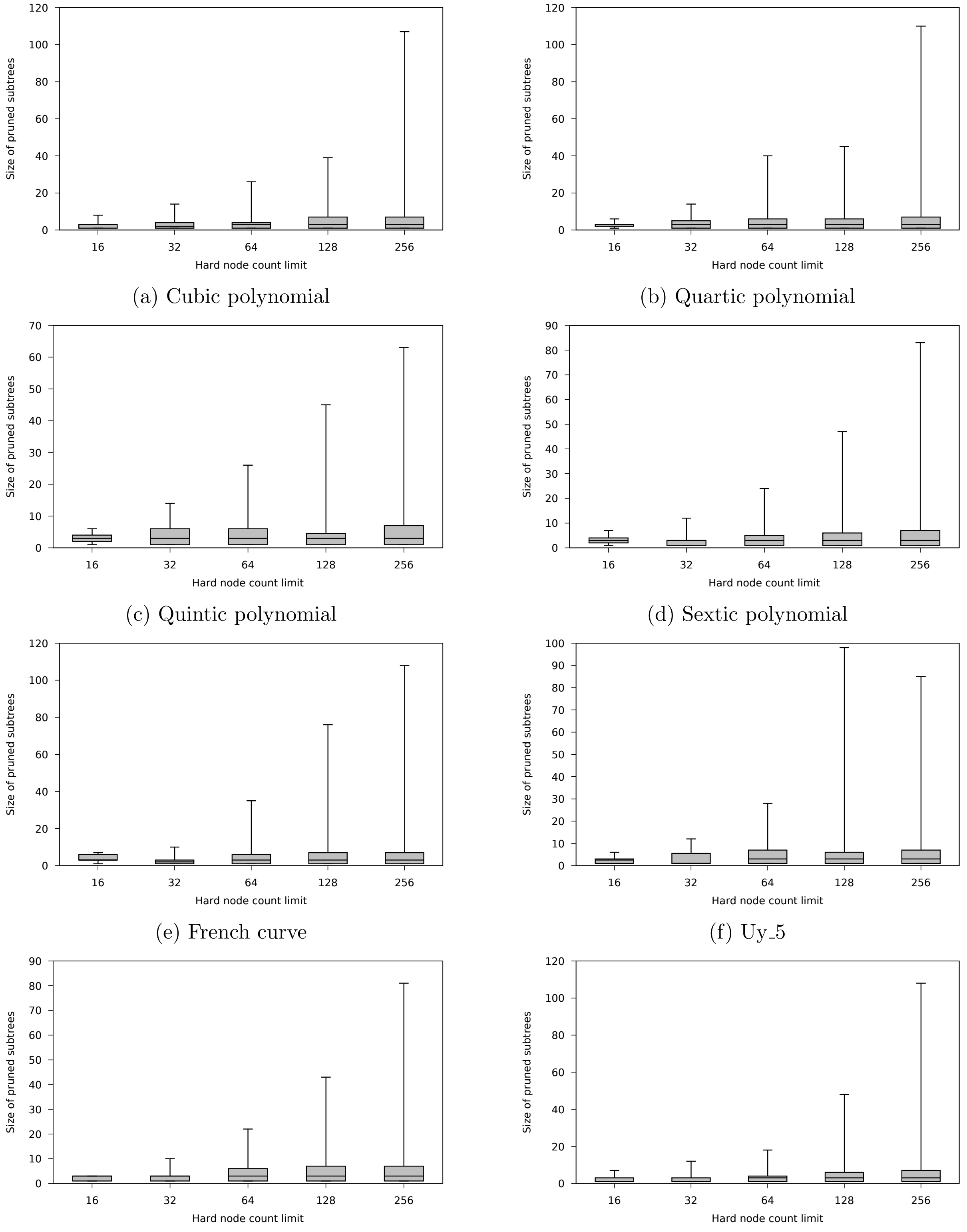
6 Discussion and Future Work

From the results presented in this report it is apparent that the proposed permutation-based pruning procedure is effective in reducing median tree sizes by about 20% independent of test function and hard node count limit. Whereas the distributions of as-evolved tree sizes clustered near to the hard node count limit, after pruning the variability of the tree sizes increased noticeably. It should also be noted that some (large) trees were pruned down to quite small tree sizes.

The results from the Salustowicz function stand alone across a number of comparisons. We have noted above the probable challenge of reliably learning a function with such a large number of extrema from just 20 data; Vladislavleva et al. [28] have previously remarked that this function is challenging for GP. Some of the trees attempting to approximate the Salustowicz function have been pruned down

Table 15: One-sided hypothesis tests for 100 training data and 20 validation data. For each hard node limit value, Z-value shows the value of the Z statistic, p the p-value of the test, and the third columns shows whether the one-sided test is for a degradation (↑) or an improvement(↓) in the generalisation error. Statistically-significant tests at the 5% confidence level are shown in bold face.

Node limit	16			32			64			128			256		
	Z-value	p		Z-value	p		Z-value	p		Z-value	p		Z-value	p	
Cubic	1.1384	0.1074	↓	1.1384	0.1074	↓	**1.5811**	**0.0253**	↓	**3.1623**	**<0.0001**	↓	**4.2058**	**<0.0001**	↓
Quartic	0.1265	0.8580	↓	**1.5495**	**0.0284**	↓	**2.0871**	**0.0032**	↓	**3.4469**	**<0.0001**	↓	**3.9528**	**<0.0001**	↓
Quintic	0.2530	0.7205	↓	0.7589	0.2831	↓	**1.6128**	**0.0226**	↓	**2.1187**	**0.0027**	↓	**3.0990**	**<0.0001**	↓
Sextic	0.0632	0.9287	↓	0.6957	0.3252	↓	1.0436	0.1400	↓	**1.8341**	**0.0095**	↓	**2.5614**	**0.0003**	↓
French	0.1581	0.8231	↓	0.3795	0.5915	↓	1.0436	0.1400	↓	**2.2452**	**0.0015**	↓	**1.4863**	**0.0356**	↓
Uy_5	0.3479	0.6228	↓	1.2017	0.0892	↓	**1.5811**	**0.0253**	↓	**2.7512**	**0.0001**	↓	**3.9528**	**<0.0001**	↓
Uy_6	0.1581	0.8231	↓	0.5060	0.4743	↓	0.9171	0.1947	↓	**1.6760**	**0.0178**	↓	**2.6563**	**0.0002**	↓
Uy_7	0.1897	0.7884	↓	1.0436	0.1400	↓	**1.7076**	**0.0157**	↓	**3.1939**	**<0.0001**	↓	**4.3956**	**<0.0001**	↓
Uy_8	0.0949	0.8933	↓	0.9487	0.1797	↓	**2.0871**	**0.0032**	↓	**3.2888**	**<0.0001**	↓	**4.3007**	**<0.0001**	↓
Salustowicz	0.0000	1.0000	↓	0.0632	0.9287	↓	0.1265	0.8580	↓	0.4427	0.5312	↓	0.3795	0.5915	↓



(a) Cubic polynomial

(b) Quartic polynomial

(c) Quintic polynomial

(d) Sextic polynomial

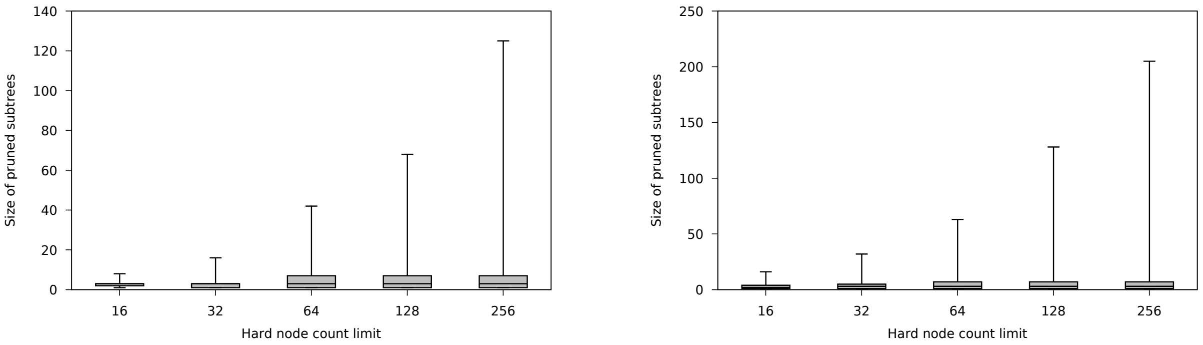
(e) French curve

(f) Uy_5

(g) Uy_6

(h) Uy_7

Figure 14: Distributions by test function of the numbers of nodes in subtrees that would have been accepted for pruning without MCP correction but were rejected with MCP.



(a) Uy_8 (b) Salustowicz

Figure 14 (cont'd): Distributions by test function of the numbers of nodes in subtrees that would have been accepted for pruning without MCP correction but were rejected with MCP.

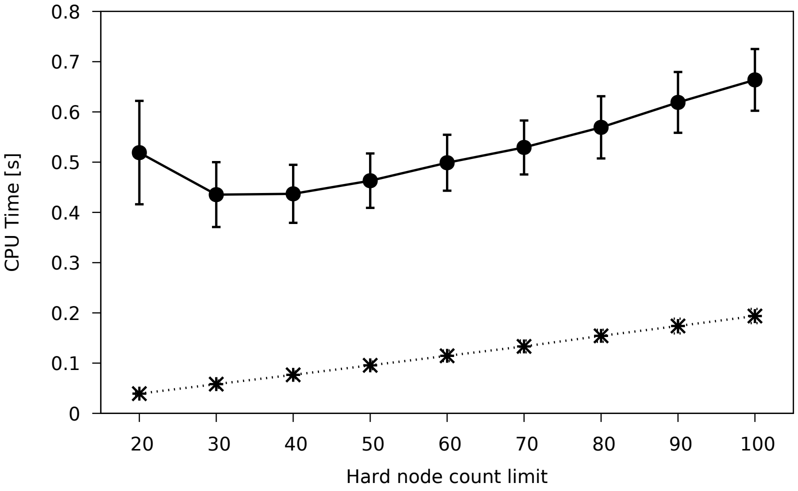


Figure 15: CPU time (in seconds) vs. hard node count limits for the French curve function. Upper, solid curve for the times taken to evolve the final population; the lower, dashed curve for the times required for pruning.

to single nodes implying that GP could find little better approximation than a constant, presumably close to the mean value of the function. Clearly, the results from the Salustowicz function flag that the the interplay between pruning and the adequacy of training warrants further research.

As regards the composition of the subtrees being pruned, around half are 3-node, binary-rooted trees having the forms: constant $\circledast$ constant, $x - x$ or $x \times 1$, where '$\circledast$' is an arbitrary binary operation. These are clearly being pruned from the peripheries of trees. Within the context of the statistically-founded method presented here, a permutation test p-value of identically zero implies (though, of course, does not prove) that a pruned subtree removal represents an algebraic simplification as opposed to an approximate simplification. The inference overall is that the overwhelming majority ($\gtrsim 95\%$) of accepted prunings are algebraic simplifications. Clearly a corollary is that only few % of prunings are approximate simplifications.

The implications for the change in generalisation error caused by pruning are interesting. A majority ($\gtrsim 58\%$) but by no means all prunings produced no change in generalisation performance for individual trees. If $\gtrsim 95\%$ of all prunings are algebraic simplifications—which, by definition, cannot change the approximating function and therefore the generalisation—then those changes in generalisation that do occur must be caused by a relatively small number of pruning events. Although we have shown that pruning does not affect generalisation *on average*, instances where test error is reduced are obviously counterbalanced by some increases in test error. In this situation, approximate prunings might be viewed as risky due to their potential adverse effects, and so a more conservative approach would be to only accept prunings with zero p-values, that is, prunings that can be inferred to be algebraic simplifications. (Notwithstanding, the original motivation of the work to allow approximate simplifications has allowed us to quantify this effect.)

Although pruning mostly leaves test error unchanged and occasionally degrades it, there are clearly occasions when pruning actually improves the test error. A similar phenomenon has been observed in the induction of conventional decision trees (DTs) where, typically, a DT is trained to the point of overfitting and then heuristically pruned to improve generalisation [29]; this phenomenon can be easily interpreted in terms of advantageously shifting the balance between goodness-of-fit and model complexity [14], pruning obviously reducing the latter quantity.

Overall, the scope for reducing tree size appears—at least in terms of the constant subtree approach used here and elsewhere—to stem almost exclusively from algebraic simplification; scope for approximate simplification appears comparatively rare and carries the risk of degrading the model generalisation although also the possible benefit of improving it. Nonetheless, even if it is only used for algebraic simplification (i.e. pruning on zero p-values), the permutation-based pruning method presented here is significantly simpler to implement than previous rule-based matching approaches.

Although 'bloat' has long been recognised as an issue in genetic programming, its definition has always been rather informal [30]. The present work possibly offers a route to a more rigorous definition in that it identifies subtrees that make no significant contribution to the program's output, serving only to increase the size of the tree. We have presented a method of judging (to within some statistical bound) whether or not a tree fragment is redundant. Although we have been concerned here exclusively with post hoc tree pruning, it is clearly an area of future work to focus explicitly on bloat.

Although we have considered only unary and binary nodes here, extension to ternary (*if–then–else*) nodes [21] is straightforward. Typically, the first subtree is used to evaluate the conditional predicate; if this predicate evaluates to 'true', then the value of the second subtree is returned, otherwise the value of the third subtree is returned to the node's parent. It is straightforward to extend the above pruning strategies to evaluate the hypotheses that the ternary node could be replaced by either the second ('true') subtree or the third ('false') subtree implying that the conditional predicate is (effectively) a constant value and that the branching structure is redundant. We leave this to future work since our work is not concerned with GP trees containing such conditional elements.

In this work, we have deliberately restricted the investigation to univariate regression problems in the hope of more easily examining the pruned subtrees to explicitly understand why they have been removed. In practice, this has proved not to be practical. Clearly future work needs to examine more complex regression functions, such as those suggested in [23]. Although the work reported in this paper is restricted to regression problems, extension to classification and indeed time series is straightforward, and will be addressed in forthcoming research.

Other variants of pruning strategy could be implemented within this permutation testing framework. For example, consider an arbitrary binary operation $z = x \circledast y$. We could examine the hypotheses that either $z \approx x$ or $z \approx y$, that is, the binary-rooted subtree could be replaced by one or other of its child subtrees [11]. This is not necessarily equivalent to $x, y \approx$ constant.

Finally, and in terms of future work on simplification, consider the expression comprising two binary nodes: $z = (x + y) - y$, which, of course simplifies to $z = x$. This expression cannot, in general, be simplified by setting any of the variables to a constant. Reductions of subtrees such as this are likely to remain a challenging task, especially if x, y are not terminal quantities but are themselves computed by (possibly large) subtrees and where y is computed by two subtrees of markedly different morphologies. Even constructing simplification rules for this case remains challenging. There thus seems a large amount of work remaining to simplify GP trees to their truly minimal form.

7 Conclusions

In this report, we have presented a novel approach based on statistical permutation tests for pruning redundant subtrees from genetic programming (GP) trees. This has the advantage of being simple to implement while not requiring the setting of user-defined tuning parameters.

We have observed that over a range of ten regression problems, median tree sizes are reduced by around 20%, largely independent of test function and the hard node count limit used to restrict tree bloat. Although some large subtrees (up to hundreds of nodes) are removed, the median pruned subtree comprises just three nodes and overwhelmingly takes the form of an exact algebraic simplification. That is, either a binary operation on two constants, a subtraction operation on two variable nodes, a multiplication of a variable by unity, or two consecutive unary minus operators and a terminal node.

The basis of our statistically-based pruning technique is that a given subtree can be replaced with a constant if this substitution results in no statistical change to the behaviour of the parent tree. This has allowed us to examine approximate redundancies where replacing a subtree with a constant produces some change, but that that change is not statistically significant. In the eventuality, we infer that $\gtrsim$95% of the pruned subtrees are the result of algebraic simplifications since this fraction of hypothesis tests yielded precisely zero p-values. These observations suggest the scope for reducing the complexity of GP trees is mostly limited to algebraic simplification and that removing approximate equivalences will have, on average, only a modest effect of pruned tree size.

The further implication of the rarity of approximate simplification prunings is that most pruning events do not change the generalisation error of the parent tree. That small number of approximate prunings that do occur, however, can have effects—both positive and negative—on generalisation.

References

[1] T. Blickle and L. Thiele, "Genetic programming and redundancy," in *Workshop on Genetic Algorithms within the Framework of Evolutionary Computation*, Saarbrücken, Germany, 1994, pp. 33–38.

[2] P. Wong and M. Zhang, "Algebraic simplification of GP programs during evolution," in *Genetic and Evolutionary Computation Conference (GECCO 2006)*, Seattle, WA, 2006, pp. 927–934.

[3] G. Dick, "Sensitivity-like analysis for feature selection in genetic programming," in *Genetic and Evolutionary Computation Conference (GECCO 2017)*, Berlin, Germany, 15-19 July 2017, pp. 401–408.

[4] W. B. Langdon, "Long-term evolution of genetic programming populations," in *Genetic and Evolutionary Computation Conference Companion*, Berlin, Germany, 15-19 July 2017, pp. 235–236.

[5] D. Jackson, "The identification and exploitation of dormancy in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 1, pp. 89–121, March 2010.

[6] M. Zhang, P. Wong, and D. Qian, "Online program simplification in genetic programming," in 6^{th} *International Conference on Simulated Evolution and Learning (SEAL 2006)*, Hefei, China, 15-18 October 2006, pp. 592–600.

[7] M. Naoki, B. McKay, N. X. Hoai, D. Essam, and S. Takeuchi, "A new method for simplifying algebraic expressions in genetic programming called equivalent decision simplification," in *10^{th} International Work-Conference on Artificial Neural Networks, (IWANN) Workshops on Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, Salamanca, Spain, 10-12 June 2009, pp. 171–178.

[8] D. Kinzett, M. Zhang, and M. Johnston, "Investigation of simplification threshold and noise level of input data in numerical simplification of genetic programs," in *IEEE Congress on Evolutionary Computation, (CEC2010*, Barcelona, Spain, 18-23 July 2010, pp. 1–8.

[9] P. Nordin, F. Francone, and W. Banzhaf, "Explicitly defined introns and destructive crossover in genetic programming," in *Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, CA, 9 July 1995, pp. 6–22.

[10] D. Kinzett, M. Johnston, and M. Zhang, "Numerical simplification for bloat control and analysis of building blocks in genetic programming," *Evolutionary Intelligence*, vol. 2, no. 4, pp. 151–168, 2009.

[11] A. Song, D. Chen, and M. Zhang, "Bloat control in genetic programming by evaluating contribution of nodes," in *Genetic and Evolutionary Computation Conference (GECCO 2009)*, Montreal, Canada, 8-12 July 2009, pp. 1893–1894.

[12] M. Johnston, T. Liddle, and M. Zhang, "A relaxed approach to simplification in genetic programming," in *13^{th} European Conference on Genetic Programming (EuroGP'10)*, Istanbul, Turkey, 7-9 April 2010, pp. 110–121.

[13] T. Helmuth, N. F. McPhee, E. R. Pantridge, and L. Spector, "Improving generalization of evolved programs through automatic simplification," in *Genetic and Evolutionary Computation Conference, (GECCO 2017)*, Berlin, Germany, 15-19 July 2017, pp. 937–944.

[14] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2^{nd} ed., New York: Springer-Verlag, 2009.

[15] P. I. Good, *Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses.* New York: Springer-Verlag, 1994.

[16] B. F. J. Manly, *Randomization, Bootstrap and Monte Carlo Methods in Biology*, 2^{nd} ed. London/New York: Chapman & Hall, 1997.

[17] M. D. Ernst, "Permutation methods: A basis for exact inference," *Statistical Science*, vol. 19, no. 4, pp. 676–685, 2004.

[18] Y. Hochberg and A. C. Tamhane, *Multiple Comparison Procedures.* New York: Wiley, 1987.

[19] S. García, A. Fernández, J. Luengo, and F. Herrera, "Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power," *Information Sciences*, vol. 180, no. 10, pp. 2044–2064, 2010.

[20] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[21] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming.* Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. [Online]. Available: http://dces.essex.ac.uk/staff/rpoli/gp-field-guide/A_Field_Guide_to_Genetic_Programming.pdf

[22] J. Ni, R. Drieberg, and P. Rockett, "The use of an analytic quotient operator in genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 1, pp. 146–152, 2013.

[23] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly, "Genetic programming needs better benchmarks," in *Genetic and Evolutionary Computation Conference (GECCO 2012)*, Philadelphia, PA, 7-11 July 2012, pp. 791–798.

[24] N. Q. Uy, N. X. Hoai, M. O'Neill, R. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: Application to real-valued symbolic regression," *Genetic Programming and Evolvable Machines*, vol. 12, no. 2, pp. 91–119, 2011.

[25] G. Wahba and S. Wold, "A completely automatic French curve: Fitting spline functions by cross validation," *Communications in Statistics*, vol. 4, no. 1, pp. 1–17, 1975.

[26] R. Salustowicz and J. Schmidhuber, "Probabilistic incremental program evolution," *Evolutionary Computation*, vol. 5, no. 2, pp. 123–141, 1997.

[27] A. Gandy, "Sequential implementation of Monte Carlo tests with uniformly bounded resampling risk," *Journal of the American Statistical Association*, vol. 104, no. 488, pp. 1504–1511, 2009.

[28] E. J. Vladislavleva, G. F. Smits, and D. den Hertog, "Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 333–349, 2009.

[29] J. R. Quinlan, *C4.5: Programs for Machine Learning.* San Mateo, CA: Morgan Kaufmann, 1993.

[30] L. Vanneschi, M. Castelli, and S. Silva, "Measuring bloat, overfitting and functional complexity in genetic programming," in *Genetic and Evolutionary Computation Conference (GECCO '10)*, Portland, OR, 7-11 July 2010, pp. 877–884.